NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

THE INSTRUMENTATION OF A PARALLEL, DISTRIBUTED DATABASE OPERATION, RETRIEVE-COMMON, FOR MERGING TWO LARGE SETS OF RECORDS

by

Gregory Alan Hammond
June, 1992

Thesis Advisor: David K. Hsiao

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DCLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Technology Curriculum Naval Postgraduate School 6b. OFFICE SYMBOL *(If Applicable)* 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS *(city, state, and ZIP code)* Monterey, CA 93943-5000	7b. ADDRESS *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 6b. OFFICE SYMBOL *(If Applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(city, state, and ZIP code)*	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
THE INSTRUMENTATION OF A PARALLEL, DISTRIBUTED DATABASE OPERATION, RETRIEVE-COMMON, FOR MERGING TWO LARGE SETS OF RECORDS

12. PERSONAL AUTHOR(S) HAMMOND GREGORY ALAN

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(year, month,day)*	15. PAGE COUNT
Master's Thesis	FROM Jan 1991 to Feb 1992	JUNE 1992	

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS *(continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUBGROUP	Distributed Database, Parallel Database, Record Merging, Multi-backend

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The Naval Postgraduate School's Laboratory for Database Systems Research is the site of the multi-backend database supercomputer (MBDS). Originally, MBDS supported a prototype primary operation (retrieve-common) which merged two sets of records in a distributed, parallel database environment. This thesis presents the testing, and modification of that prototyped primary operation.

First, the design rationale of the MBDS is reviewed. Specifically, this review examines the reasons for a database-oriented supercomputer, the MBDS primary processes, and the methodology of distributing a database within loosely coupled and highly parallel database stores. Then, this study explains the methodology involved in developing theories on the cause of retrieve-common's defects and bottlenecks. Finally, in validating our theories, this study relates the process of discovering and correcting these discrepancies.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL DAVID K. HSIAO	22b. TELEPHONE (Include Area Code) (408) 646-2253	22c. OFFICE SYMBOL CS/Hq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited.

The Instrumentation Of A Parallel, Distributed Database Operation, Retrieve-Common, for Merging Two Large Sets Of Records

by

Gregory Alan Hammond
Lieutenant, United States Navy
B.S., California State University, 1983

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1992

ABSTRACT

The Naval Postgraduate School's Laboratory for Database Systems Research is the site of the multi-backend database supercomputer (MBDS). Originally, MBDS supported a prototype primary operation (retrieve-common) which merged two sets of records in a distributed, parallel database environment. This thesis presents the testing, and modification of that prototyped primary operation.

First, the design rationale of the MBDS is reviewed. Specifically, this review examines the reasons for a database-oriented supercomputer, the MBDS primary processes, and the methodology of distributing a database within loosely coupled and highly parallel database stores. Then, this study explains the methodology involved in developing theories on the cause of retrieve-common's defects and bottlenecks. Finally, in validating our theories, this study relates the process of discovering and correcting these discrepancies.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

Two people made significant contributions to this thesis, Dr. David K. Hsiao (Professor, Naval Postgraduate School) and John Locke (Systems Programmer, Naval Postgraduate School). It was their congeniality, helpful suggestions, and assistance during critical points of development which allowed this thesis to be completed successfully.

I. AN INTRODUCTION TO A SUPERCOMPUTER-DATABASE MACHINE

The increasing desire to access and manipulate greater amounts of complex information has led researchers to search for methods of improving the performance of the Database Management System (DBMS). An area that shows increasing promise is a DBMS that can perform operations in parallel.

A. SUPERCOMPUTERS FOR NUMERICAL COMPUTATIONS

The use of parallel operations in a conventional supercomputer for speeding up computations is not new. There are many production-level, numerical-oriented supercomputers. However, these types of supercomputers are not effective with operations that involve database structures. Lazou [Ref. 1] concurred with our observation by stating that conventional supercomputers are designed for maximizing speeds in calculating floating-point numbers. To fulfill the requirement of fast computations, these types of supercomputers have been specifically designed with a multiplicity of scalar or vector functional units and CPUs. They are designed to receive operands and deliver results under parallel conditions. The capabilities of these scalar or vector functional units are limited, since they are restricted to numerical operations only. This limitation to numerical operations means the database operations will not be able to take advantage of the parallel processing capability of the conventional supercomputer.

In addition to the limited capabilities of the functional units, the CPUs are not effective for database operations either. Very few database problems fall within the characteristics that take advantage of multiple CPUs of a numerical

supercomputer. Specifically, a conventional supercomputer's CPUs require a computational problem to be sectioned into small and parallel portions. Standard database operations (e.g., retrieve and update) cannot be divided into small and parallel portions for numerical processing, since database operations are mostly non-numerical.

B. SUPERCOMPUTERS FOR DATABASE MANAGEMENT

The supercomputer designed to provide parallel operations for a DBMS can be found in the Multiple Backend Database Supercomputer (MBDS). As a prototype system, the MBDS is developed to provide the necessary architecture for performance gains and capacity growth via parallel database operations. Performance gains for the same transaction are obtained by increasing the degree of parallelism in database management. Capacity growths may be facilitated for the same response time, if the degree of parallelism is proportional to the database growth.

MBDS utilizes dedicated computers (called database backends) configured from multiple, identical, and off-the-shelf microcomputers, each of which has its own external storage devices. The architecture of MBDS is illustrated in Figure 1.

The architecture illustrated in Figure 1 is scalable because it introduces parallel backends and their stores in proportion to the performance gains and capacity growth desired. More precisely, this architecture allows system processes to be replicated onto new and additional backend computers. These replications allow parallel processing of database transactions and parallel accesses to the database.

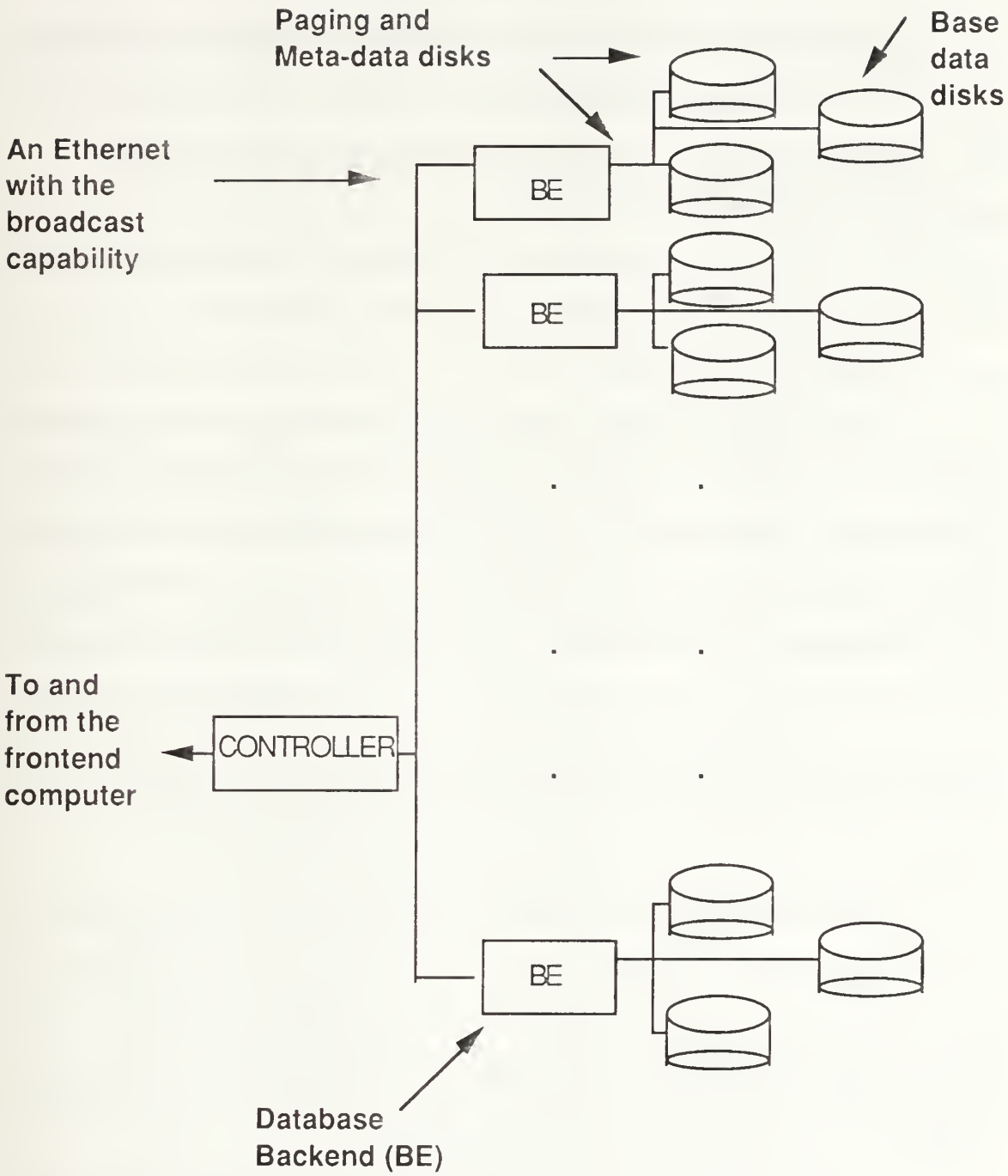


Figure 1. The Multibackend Database Supercomputer

These parallelisms of MBDS have been shown to improve the performance of DBMS substantially and proportionally.

C. THE PROCESSES OF THE MULTIBACKEND DATABASE SUPERCOMPUTER SYSTEM

MBDS software (i.e., processes) functions are discussed in two major subsections: the controller subsection and the backend subsection.

1. Controller Processes

The controller computer supports five main processes which direct the operation of the controller computer. These processes are known as Request or Transaction Processing (TP), Post Processing (PP), Insert-Information-Generator (IIG), Put, and Get. TP interfaces with the user, identifies the user and pre-processes each transaction. Specifically, each transaction is parsed, checked for syntax errors, and formatted. Upon completion of this pre-processing, TP broadcasts the transaction to all of the backends which in turn store the incoming transaction in their respective transaction queues. PP also interfaces with the user. It provides transaction results to the user.

To ensure that each transaction is returned to the correct user, PP maintains the ability to interact with TP to match transaction responses to appropriate users. Additionally, PP performs aggregate functions on data returned from the backends. For example, summations and averaging are conducted on the data that have been provided to PP.

Get and Put provide the controller with the capability to communicate via the Ethernet to the processes residing on the backend computers. Specifically, Get allows the receipt of information from the backends. When

communicating with the backends, Put allows the transmission of information in the one-to-one or one-to-many, i.e., broadcasting mode.

Finally, IIG is considered a critical process of the controller. This process is responsible for the even placement of record clusters into the database stores of the backends. The concept and importance of the record cluster will be elaborated on in a later section. Here, we consider it simply as a record set. IIG first determines the backend into which a record is to be inserted. This determination is completed by using the space utilization table which maintains the disk-track information of all the backends' base-data disks. When an appropriate track is determined, IIG directs the loading of records into the track of a backend. Following the insertion, IIG directs the updating of the tables in the meta-data disks as required. IIG's space utilization table provides the following information:

a. It identifies the backends that contain the first and last trackful of records of a particular cluster.

b. It identifies the backends that can provide new tracks for new records of a cluster.

2. Backend Processes

In a backend computer, there are five processes that direct all the backend operations. These processes are Directory Management (DM), Record Processing (RP), Concurrency Control (CC), Get, and Put.

DM is responsible for managing and accessing meta-data, i.e., contains information about base data For example, a descriptor has the value range of a particular attribute in the base data. Upon the receipt of a query of a transaction from TP, DM in each backend takes the keywords of the query, and searches the

meta-data store for the matching descriptors. When the appropriate descriptors are located, it determines the clusters (if any) to which the records belong. This information is then transmitted to RP.

RP is responsible for the access and manipulation of records. Specifically, RP performs record retrieval, selection (based on additional attribute-value pairs of the query), and the extraction of attribute values. Therefore, it is intricately involved with the disk input/output operations.

CC is responsible for maintaining meta-data and base-data integrity during the execution of user requests or transactions. Because the data requirements of user requests may overlap, it is important that the data consistency is maintained while requests are being processed. There is no CC function in the controller because all of the user requests are fulfilled by the backends. Here, Get and Put provide the same communication capabilities as Get and Put of the controller. Figure 2 illustrates the relationship of the controller processes and the backend processes.

D. THE CLUSTERS OF THE MBDS DATABASE

The replication of DBMS functions onto independent and parallel backends is the first step in providing parallel operations for a multiuser DBMS. The second step is related to the accessibility of the database stores. In a conventional DBMS, accesses are always made to a common database store. This mode of accesses is considered adverse to parallel operations. However, the adversity of accessing a common database store is directly related to the system's requirements to maintain data consistency.

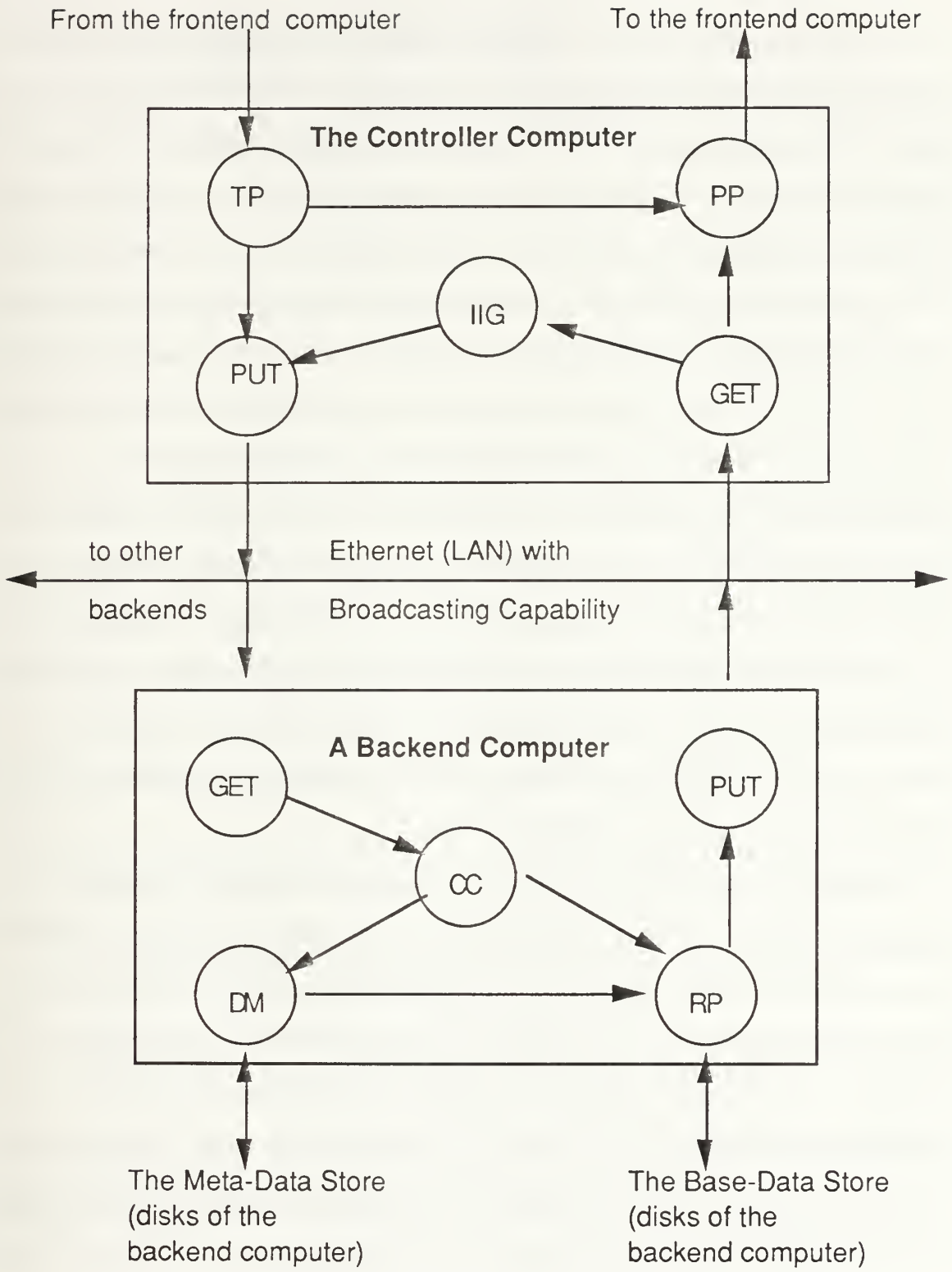


Figure 2. The Organization of MBDS Processes

In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by user transactions. These user transactions retrieve and modify data that is present in that database store. They can be executed concurrently and may access and update the same database. If this concurrent execution is not controlled, it may lead to an inconsistent database, i.e., a database with incorrect information [Ref. 2]. A technique to control concurrent executions of transactions is based on the locking concept. Elmasri [Ref. 2] defines a lock as a variable associated with a data item in the database. This variable describes the status of that data item with respect to possible operations that can be applied to it. Essentially, read locks allow transactions that do not modify the data to have accesses with other transactions involved with reading only. However, transactions that are involved with writing can only have accesses to data if no read or write locks exist over the data. The write locks do not allow any other transactions to have any access to the data. In general, the locking mechanism ensures that the integrity of the database store is maintained by controlling accesses to the store.

Locking is just one of the many concurrency control methods; however, it highlights the adverse characteristic of using a common database store. If MBDS were to utilize a common database store, the backends would experience delays due to being locked out of information in the common store necessary to complete a transaction. Therefore, performance gains by using multiple and parallel computers would be nullified. The solution to this obstacle is to develop a method that would evenly distribute (partition) the contents of "the common database" to the multiple database stores - one for each backend.

1. The Partitioning of the Database

A **Partition** of a set **A** consists of the subdivision of **A** into a collection of subsets which are pair-wise disjoint and whose union is **A**. The use of partitions ensures that each backend performs its operations on a unique subset of the database on its own database store. Therefore, the parallelism may be maintained without performance degradation, since there is no contention over a single common store. Instead, all the parallel operations are performed on their database partitions parallelly.

The technique used to partition the records is based on the notion of an **equivalence relation.** The ideal behind an equivalence relation is that it is a classification of objects which are in some way "alike." The formal definition of an equivalence relation [Ref. 3] is as follows: A relation on a set is an equivalence relation if it is reflexive, symmetric, and transitive on elements of the set.

The properties of reflexive, symmetric and transitive is presented below for the set F where the relationship is represented by the symbol $\&$.

a. The relation $\&$ is reflexive. If for each a that is a member of F, the following is true: $a \,\&\, a$.

b. The relation $\&$ is symmetric. If for each a and b that are members of F, the following is true: $a \,\&\, b$ implies $b \,\&\, a$.

c. The relation $\&$ is transitive. If for each a , b, and c that are members of F, the following is true: $a \,\&\, b$ and $b \,\&\, c$ implies $a \,\&\, c$.

An abstract example presenting cases where a relationship does not fulfill the equivalence-relation requirements (transitive, reflexive and symmetric) is presented below:

Consider the relation **TT** = {(1,1), (1,2), (2,1), (2,3)} on the set **A** = {1,2,3}.

a. Both 1 and 2 are members of **A**; however, (2,2) is not a member of the relationship set **TT**, although (1,1) is in **TT**. Therefore **TT** is not reflexive.

Since a relation must be symmetric, transitive and reflexive to be an equivalence relation, **TT** is not an equivalence relation.

The notion of equivalence relations is used because it allows us to broaden the notion of equality from identity. Elements are judged on similarity based on being alike relative to a common property. As stated in [Ref. 3] " two elements need not be identical to be equivalent; they need only to share a specified property." This sharing of a specific property allows us to explain the interrelationship of equivalence relations, equivalence classes, and partitions.

The formal definition of an equivalence class [Ref. 3] is as follows: "Let $\sim$ be an equivalence relation on a set A. For each a that is a member of A, the equivalence class of a is the subset, denoted by $[a]$, consisting of all elements x of A that are equivalent to a , i. e. , $x \sim a$" This definition allows us to review a theorem provided in [Ref. 3] which presents the basic properties among elements of an equivalence relation. Specifically, the theorem assumes that $\sim$ is an equivalence relation on a set A and that elements x, y are members of A, the following rules apply to $\sim$:

a. If $x \sim y$ is true, then $[x] = [y]$.

b. If *not* $(x \sim y)$ is true, then the intersection of $[x]$ and $[y]$ is empty.

c. The union of all the equivalence classes of $\sim$ is A.

The interrelationship of partitions and equivalence relations becomes evident when we invoke the aspect of equivalence classes. The rules of equivalence classes indicates that for any equivalence relation $\sim$ on a set A, the

set of distinct equivalence classes of A modulo $\sim$ constitutes a partitioning of A. This stipulates that for every equivalence relation on a set A, there exist a corresponding partition of A in terms of those equivalence classes [Ref. 3].

2. The Distribution of a MBDS Database

The determination, that (1) equivalence classes develop database partitions and that (2) the union of these database partitions provide the whole database, is the foundation of our database distribution methodology. The distribution methodology develops similarities by using common attributes and the attribute-value ranges of the records within the database. These attributes and ranges are used to develop an equivalence relation and its corresponding equivalence classes. The equivalence classes develop mutually exclusive partitions (called clusters in MBDS). These clusters allow the even distribution of a database onto the backends' stores of MBDS.

The clusters are distributed onto the backends based on an one-track-per-backend-store algorithm. A cluster of records are inserted onto a backend's database store (disks) until the track is full. When it cannot receive any more data, then another backend's database store is selected to receive the next track of the clustered data. For example, if a track on the database store of backend number three is full, then the database store of backend number four will be selected to receive the next track of clustered data. The algorithm, which is embedded in the IIG process, determines the next database store of a backend modulo the number of backends. Figure 3 illustrates the distributing of the records to the database stores, i.e., external storage devices.

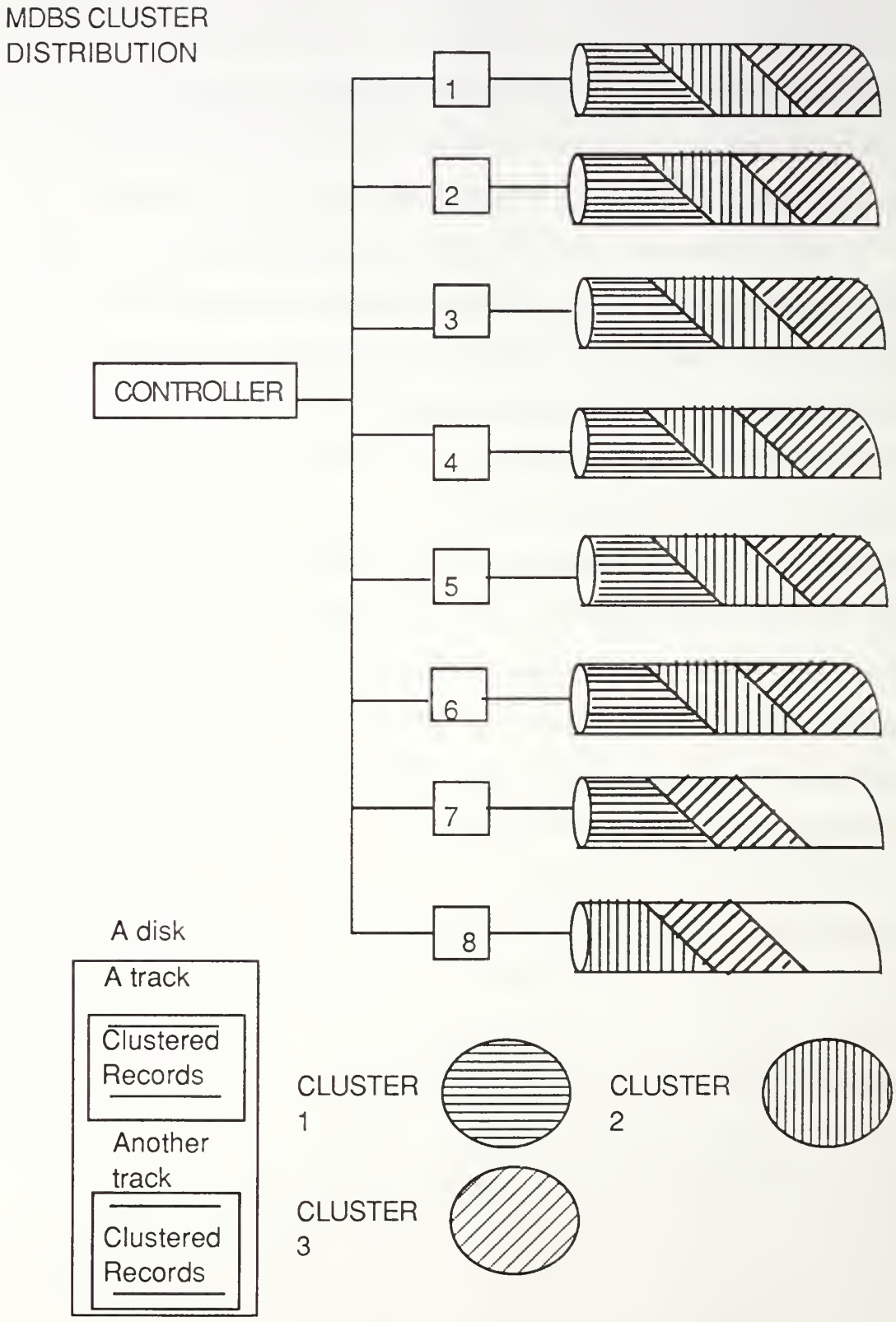


Figure 3. MBDS Distribution Strategy

The development of the method to evenly distribute clustered records into the datastore allows the extensive and scalable architecture of Figure 1 to be effective. The MBDS allows every backend to process the same transaction simultaneously. Each backend only needs to know the base data contained in its database store. This architecture is the foundation of the MBDS parallel processing capability; which incurs no delays and no lockouts in parallel accesses to the commonly clustered database.

E. THE MBDS PRIMARY OPERATIONS

There are five primary database operations in MBDS. They are Retrieve, Delete, Update, Insert and Retrieve-Common. The primary operations, Retrieve, Update and Delete, operate on a set of records at a time, while Insert operates on a single record at a time. The retrieve-common primary operation is different from other primary operations. It manipulates two sets of records. This manipulation of two sets of records leads to the uniqueness of the primary operation. Each of these sets of records is determined by an independent query. These distinct sets of records are then merged on the basis of a common set of attributes values specified by the user. In Figure 4, we present a sample retrieve-common transaction for illustration.

This sample retrieve-common will merge census records with common names of U. S. cities and Canadian towns. The output would be the names of the city or town and their respective population figures.

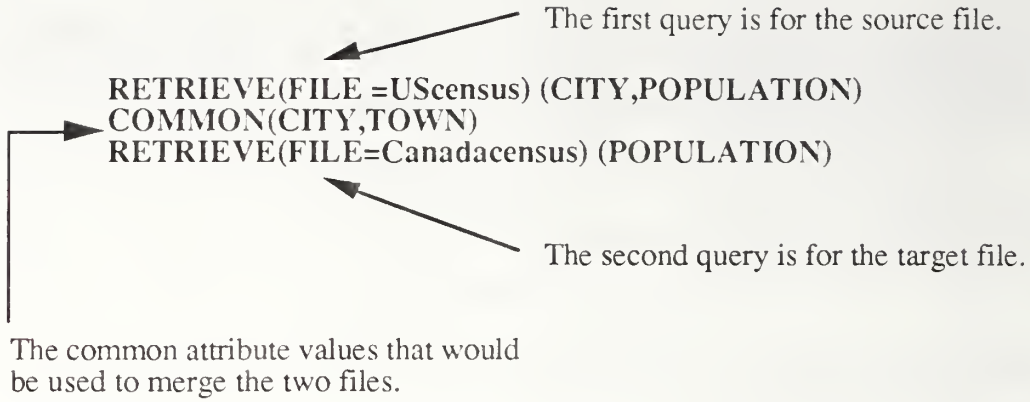


Figure 4. A Sample Retrieve-Common Transaction.

1. The Comparison of The Retrieve-Common and Equi-Join

The retrieve-common primary operation is equivalent to the relational equi-join operation. However, differences do exist. pecifically, an equi-join manipulates two sets of relations in a single DBMS with only one computer [Ref. 2]. When the appropriate tuples of these relations are collected, they are merged into a new relation. This new relation is then provided to the user as the result of the user's query. A retrieve-common, however, is designed to operate in a parallel DBMS on an incrementable number of backend computers. Specifically, while conducting such an operation, clustered records on each backend are being searched for records whose attribute value pairs fulfill the user query. When that search is completed, however, the backends cannot consider the user's query to be satisfied merely by merging the appropriate records on common attribute pairs. As highlighted in our discussion of the database-store distribution, each backend only contains a partition (subset) of the database. Therefore, to ensure that an adequate merge of attribute values pairs does occur, the retrieve-common allows backends to share their individual partitioned data. This provision is accomplished by the transmission of one's

partitioned data to other backends. Provisions of the equivalence classes ensure that the sharing of partitioned data (i.e., clustered data) in this manner maintains the integrity of the database partition (or cluster). All appropriate attribute value pairs will be reviewed before a final result is provided to the user. The reliance on the notion of the equivalence classes, the subdivision of the database into partitions, and retrieval and sharing of partitioned data from individual backends is an intricate element in the design of the retrieve-common. Without these capabilities, MBDS will not be able to conduct parallel merges.

Due to its operational complexity and parallel nature, the retrieve-common's coordination, communication and query processing requirements exceed the requirements of an equi-join.

2. The Retrieve-Common Algorithm

The algorithm is provided in the single-query-multiple-data-stream mode as follows:

a. The controller will broadcast the retrieve-common transaction to all the backends to be inserted into their respective transaction queues.

b. For that transaction, each backend will retrieve its first set of clustered records (called **source records**) from the first query of that transaction.

c. For each record retrieved, each backend would hash the record into its virtual memory based on the common attribute value of the record This process would continue until all of the retrieved records are hashed into its virtual memory.

d. Each backend will now retrieve the second set of clustered records (called target records) that fulfill the second query of the transaction.

e. For each of these target records retrieved, the common attribute value is hashed to provide a virtual memory address. At that point, the records of that virtual memory address are fetched one by one and compared against

this record. If they do compare, then they are merged and prepared for output.(see step h.). This process continues until all records of the second set have been retrieved, compared, and processed.

f. Each backend then broadcasts its second set of clustered records to all the other backends.

g. For each record received via broadcasting, each of the backends will repeat step e. The process of broadcasting target records to the other backends will continue until a flag indicating completion is received.

h. Finally, each backend will merge their source records (which met the first query) with the target records (which met the second query) and outputs the results to the controller.

F. THE AIM AND INTENT OF THE THESIS

The preceding introduction of the architecture and design rationale of MBDS allows us to state the aim and scope of this thesis. Presently, the implementation of retrieve-common is defective. It only allows the manipulation of a small database. When the database reaches a size that is appropriate for reasonable database operations, MBDS fails. Before the completion of this thesis the cause of this failure was unknown.

The aim of this thesis is to develop a theory to explain the cause of the defective retrieve-common operation and to correct the defect. The thesis will determine whether the defective operation is the result of architectural deficiencies, inadequate hardware support, a defective algorithm, or erroneous implementation. When such deficiencies are identified, this thesis will present the appropriate correction. The final intent of this thesis is to provide a methodology to troubleshoot (debug) very large parallel systems. The increasing importance of conducting parallel operations accentuates the necessity of having an effective methodology for debugging parallel operations.

G. THE ORGANIZATION OF THE THESIS

The remaining parts of the thesis are organized as below:

Chapter II evaluates whether or not architectural deficiencies exist in the present implementation of the retrieve-common. The results of that evaluation can direct the development of theories regarding the cause(s) of the defective retrieve-common operation. Chapter III discusses the documentation which has been developed to appropriately evaluate (i.e., debug) a complex parallel-backend,multiprocess-based system such as MBDS. Additionally, Chapter III determines which of the defect theories have merit and presents corrections that have been implemented to resolve those defects. Chapter IV presents our findings, and provides directions towards future research.

II. THE DEVELOPMENT OF THEORIES OF DEFECTS

A. A STUDY OF HARDWARE LIMITATIONS AND SOFTWARE ALGORITHMS

Early research indicates that three methods were proposed for implementing the retrieve-common in MBDS [Ref. 4]. The primary consideration behind each of these methods involves the location for the merging of two sets of retrieved data. The methods are reviewed briefly here:

Method 1. The controller does the entire merge operation.

Method 2. The controller and the backends share the workload of the merge.

Method 3. The backends do the entire merge operation.

The first and second methods were discounted because they violated the major design goal of MBDS: to minimize the work and involvement of the controller. The designer believes that by minimizing the controller interaction (a) greater levels of parallel operations by the backends are possible and (b) less likely that the controller will cause a bottleneck. Since more activities can be completed parallelly in the individual backends, there is no need to do them serially in the controller. Additionally, allowing the controller to complete the merge operation can provide the possibility of a bottleneck at the controller. This bottleneck can result in two ways: through the transmissions from the various backends, and from the interactions with the frontend computer.

Thus, the first two methods were eliminated. Method three is the basis for the design and implementation of retrieve-common that is presented in Chapter I which does not have the limitation of either method 1 or 2 as articulated above.

The defective performance of retrieve-common generates doubts about the merit of the backend-based method three. Theoretically, the system architecture in Figure 1 is sufficient for completing the backend based merge operation [Ref. 5]. However, the system's inability to manipulate large amounts of data from database stores in retrieve-common provides a justification for review of the system hardware performance under aforementioned methods. We hypothesize that the hardware limitation of the backends could reduce the performance of the backend-based merge operation, i.e., method three. On the other hand, the controller bottleneck discussed earlier in the controller-based merges may have smaller ramifications than anticipated. We also consider the possibility that the hardware used to implement the primary operation may include restrictions for parallel processing. These restrictions may favor the controller-based implementation of retrieve-common, since it is a serial processor.

The hypothesis that hardware limitations may invalidate the merit of the backend-based merge, i.e., method three, has been found to be untrue. The hardware characteristics of the MBDS system [Ref. 6] do not provide performance restrictions on method three. Based on kernal program results, we observe that the backend-based merge outperforms the controller based merge by about 60 percent. Additionally, we observed that the present algorithm is implemented according to the designer's specifications.

Our determination that the backend-based retrieve-common algorithm is not effected negatively by the present hardware elements of MBDS allows us then to review the software implementation.

B. TOWARDS THEORIES OF DEBUGGING

Since the retrieve-common algorithm utilizes a number of system processes, a thorough understanding of the individual processes as well as their interrelationships is necessary. The interrelationship of the major processes ensures that any modification to one will affect the other system processes accordingly. Modifications are not restricted. But, a thorough understanding of the processes and their interrelationship is required prior to any attempt to determine and correct implementation errors. Without this understanding, we may fail to determine the deficiency and make the corrections.

1. Conducting Test Runs

The first step is to develope a theory regarding the deficiency of retrieve-common and the interrelationship of system processes by conducting test runs of the MBDS system. The test runs indicate that the MBDS system operates for all five primary operations. Moreover, the retrieve-common performs incorrectly only beyond certain amounts of retrieved data from the database stores. An initial hypothesis is ascertained from these tests. We conclude that the basic logic, i.e., the algorithm of the primary operation must be correct. If the basic logic is incorrect, the tests will not operate correctly under any condition. We then infer that the problem with retrieve-common must be related to the defective implementation of some data structures or functions for the algorithm. However, these data structures and functions are shared among several system processes. Any change will affect the interrelationship of the

system processes. Additionally, the primary operations use other primary operations for its own operation For example, the retrieve-common uses the primary operation, Retrieve, twice to obtain the first and second set of records, i.e., source and target files from the database stores. These records are then manipulated by retrieve-common in order to provide the correct result.

2. Placing Debugging Flags

The complexity of process interrelationships in MBDS requires us to narrow our focus on the problem area quickly. This is achieved by using compilable debugging flags to determine which processes have been involved in the primary operation, retrieve-common. These flags provide information regarding the variables passed, and messages sent by these involved processes.

The use of these compilable flags is also instrumental in determining the sequence in which various processes and primary operations are used to complete their assigned tasks. Once the debugging flags have been compiled in place, a retrieve-common test run is initiated with a database size that is known to allow the operation to complete correctly. This test run allows us to identify all the functions, processes, and programs involved.

3. Identifying File Locations

The flags are not capable of indicating the locations of the files in which these functions, processes, and programs are stored. And since there are over 100 such files for MBDS, this limitation must be overcome.

The search mechanism in the operating system is ineffective, because the MBDS file structure is formatted in several layers of abstractions. These layers of abstractions require that a search request is implemented at a specific layer in order to obtain the correct result. We observe that documentation tools

are needed to allow the determination of file and function information more efficiently. In a later chapter these documentation tools will be described.

4. Determining the Threshold of failure

The next step is to initiate the retrieve-common with a database large enough to cause the primary operation to fail. Since this database size is not known, numerous operational tests are required. The operation fails when it is operated on a database of 45 records with an average size of 32 bytes per record.

Before the system fails, it provides a trace of processes and functions that have been entered and exited via debugging flags.

5. Using Error Feedbacks

Wherever there is an abnormal shutdown of MBDS, a pool of error indicators is presented in the error-feedback system of MBDS. The error-feedback system provides an outlet for error indicators and messages from the operating system and MBDS. It consists of six permanent files. Each is assigned to a process of the MBDS. When MBDS is running, these files allow for the insertion of debugging data, error indicators, and diagnostic messages. A number of such data, indicators and messages are discussed herein. The first type of error message in the feedback system is usually of a message-header error. The message-header error indicates that somewhere in the system a message is sent with a defective message-header. The defective message-header has caused the message to be undeliverable and initiated the operating system to suspend the message-sending processes. Once the running process is suspended, the operating system generates the error message that has been placed in the appropriate file for the process. This type of error message is termed ***illegal***

ioctrl. After reviewing it, we determine that this type of error is sufficient to cause the MBDS system to experience an abnormal system shutdown.

Another type of error indicator is also caused by the defective retrieve-common. This indicator suggests that system malfunctions have occurred outside of the system. One indicator, ***bus error***, for example, may be due to too many processes being concurrently executed by the operating system. Although the **Berkeley 4.3 Unix Operating System** has the ability to conduct concurrent processing [Ref. 6], there is a limit on the number of processes the operating system can manipulate concurrently. The ***bus error*** can imply that this limit has been reached and that the operating system needs to notify the user. The operating system then suspends all running processes, places the error message in the appropriate file, and directs the abnormal shutdown of the MBDS system.

Consider a third type of error caused by the defective retrieve-common, the ***write error***. This error message usually indicates that the system has attempted to write to an external storage device that is full or not available. For writing, the operating system provides an interface between the disk and the user as shown in the five steps below [Ref. 6]:

a. The operating system allocates a buffer to accept the data provided by the user or user process.

b. The operating system determines a location on the external storage device to place the information as indicated by the user or user process.

c. The operating system requests the controller of the external storage device to read the contents of the physical block into the system buffer.

d. The operating system copies the contents in the input/output buffer of the user or user process to the appropriate portion of the system buffer.

e. Finally, it writes the system-buffer block back to the external storage device.

The *write error* indicates that there is an error in one of the preceding steps. As with other errors already mentioned, this error will cause the operating system to terminate the system processes of MBDS.

The myriad of errors has compounded our search for the cause or causes of the defective retrieve-common. The dissimilarity of these errors have not related them to one particular problem. Additionally, because each of the errors has caused the system to terminate abnormally, the cause of that termination could not be traced in real-time to a single function or process.

C. SIX THEORIES ON DEFECTS

The inability of error messages to direct us to a definitive system defect has led to the development of separate theories based on the available information on hand; which includes usage patterns, test results, debugging flags, and error messages. Individually, these factors could not provide any assistance; however, when combined some portions of the problem, they may become visible. The culmination of debugging information allows us to develop six plausible theories regarding the defective operation of the retrieve-common. Two of these theories are related to the communication aspects of the MBDS system; three theories are related to data manipulation by MBDS; the last one is related to the operating system. These theories are presented below:

1. Defects in Communication

The retrieve-common requires processor communications in broadcast mode. This mode of communications has resulted in many message-header

errors which leads us to propose the possibility of two communication related errors:

a. There may be a MBDS design limitation on the size of the message being broadcasted. Therefore, the system fails if the size of the message grows beyond the limit.

b. An operating-system-interface problem may exist. The retrieve-common may require different sockets to be utilized during different activities, thus causing the possibility of a socket-related error. The socket-related error would provide a header error from the operating system..

2. Defects in System Processes

Since the write errors point to possible defective interfaces, the problem area may be narrowed by initially reviewing the following:

a. PP (i.e., the postprocessing process) for the output combined records of the retrieve-common in the controller computer.

b. The disk I/O process for base data (i. e., both the source and target files) in retrieve-common's record-processing process.

c. The hashing process for storing the source file of the retrieve-common in virtual memory temporarily.

3. Defects in Operating System Supports

As discussed earlier in the chapter, a bus error is related to the number of active processes in the operating system. The possibility that the number of active processes surpassing the limit designed into the operating system is small.

D. THE STRATEGY FOR EVALUATING THE THEORIES

The capability of the system to operate correctly with very small databases marks the possibility of a defect in MBDS processes. The three theories of defects in system processes are therefore pursued first. The broadcast communications are built on the protocols of the Ethernet. They are the next place to look for defects. Thus, the two theories on communications defects are considered next.

Operating system-related errors are the least plausible. The ability of retrieve-common to spawn an abnormal number of processes is very small. Therefore, this theory is to be researched last. In this way, the theories with the most promising defect detection and corrections ideas are applied to the problem first.

III. DETECTIONS AND CORRECTIONS OF DEFECTS

In Chapter II we have developed various theories for the possible defect in retrieve-common. We now apply these theories to the detection and correction of defects found in the retrieve-common.

A. A REDUCTION OF THE NUMBER OF PROCESSES TO BE ANALYZED

All of the error indicators resulting from our testing enable us to conclude that certain parts of the system are operating correctly. Therefore, we are able to reduce the number of processes that may have defective operations. Specifically, with the exception of the communication and record processing processes (i.e., GET, PUT, and RP), we conclude that all of the other backend processes are operating correctly. Since the directory-management and concurrency-control processes (i.e., DM, and CC) are operating correctly during the primary operations of inserts, deletes, and retrieves, they should continue to operate correctly in supporting the retrieve-common.

We also tested the controller processes. We are able to conclude that the insert-information-generator and the request-processing processes are operating correctly (IIG and TP). Specifically, in IIG the placement of clustered records in the database stores is being conducted correctly; TP is operating correctly for all other primary operations where all requests are properly identified, formatted and transmitted correctly.

Nevertheless, we must examine the five processes TP, RP, PP, GET and PUT more thoroughly, since they support the retrieve-common operation. We

run the identical retrieve-common with two different database sizes one of which causes a system failure. This test indicates that the identical primary operation formatted by TP has operated correctly on a smaller database size only. Thus, this test provides the necessary evidence that the formatted request provided by TP may not be a factor to the system failure. Perhaps, the system has failed due to other factors contributed by other processes in handling the larger database size.

Some other controller processes can not be discounted as error-free. For instance, there is some evidence from the error indicators that a possible defect may exist in the communication processes, Get and Put, which are to be discussed in this chapter. As the larger size of the database effected the system performance, the handling of the large amount of results by PP may be the cause of errors too. Finally, the backend process RP which accesses individual records of a large database has shown many error indications. We should examine it thoroughly in the context of large database sizes.

Of the five processes we have mentioned above, four may cause the retrieve-common to be defective. These four are PP, RP, GET and PUT; their testing and evaluation in the context of large databases are presented in the later sections of this chapter.

B. THE IDENTIFICATION OF DOCUMENTATION REQUIREMENTS FOR DEBUGGING

In maintaining and debugging a complex system such as MBDS, the system documentation is critical. Effective documentation assists in the efficient determination of how a given process performs. Additionally, with the documentation, modifications can be made to the process at appropriate places.

The documentation that is necessary to evaluate the MBDS processes can be considered at three levels of detail:

a. **Process Map** - This documentation is developed for each of the system processes (RP, CC, DM, etc.). It provides a high-level view of what events are accomplished and when a particular process is activated. It presents which procedures are called, what purposes are intended, and where files of the source code are located.

b. **Process Pseudo-Code** - This documentation is also developed for each of the system processes. It provides a short description of the tasks completed by those procedures which have been highlighted in the Process Map. The Process Pseudo-code does not provide detailed information on how procedures complete their tasks.

c. **Transaction Flow** - This document explains the events involved with specific procedures, and a detailed transaction flow is developed. This transaction flow represents the succession of events involved in a particular subprocess or procedure. This documentation is presented in flowcharts, which illustrate the logic of a specific procedure.

Appendices A, B, and C provide excerpts of the above three levels of documentation. These excerpts should be used as a documentation guide for system developers. The availability of three levels of documentation allows system users and staff to select the level of documentation they require to complete there task.

C. ASSESSMENTS OF THEORIES OF DEFECTS

With three levels of documentation, we now proceed to apply our theories of defects to the detection and correction of the retrieve-common operation.

1. Communication-Related Theories of Defects

In Chapter II, we have presented two communication-related theories of defects. The first theory suggests that messages in the transmission during retrieve-common may be limited in size. The defective performance that occurs

at larger database sizes may be related to an inability of GET or PUT to handle messages after these messages have surpassed a fixed message size. To validate this theory a review of the message structures involved in transmitting messages in the retrieve-common is conducted.

The primary operation, retrieve-common, transmits and receives only one message (**BucketInfo**) specific to the operation. This message delivers the target records of a particular backend to the other backends. The **BucketInfo message** is a formatted message that uses a fixed header. The header is computed and formed during the insertion of records into the message buffer, i. e., the message development. While reviewing the message development, we note that the record addresses in the header are static and not modifiable. Each backend transmits its **BucketInfo message** with the same header format. The format of this message is presented in Appendix D.

Now, we apply our first theory of communication-related defects. Specifically, the theory is that a message routing error is caused by the header error of the message. A routing error could only occur if the message transmitted by retrieve-common uses a variable format for its addresses.

Since the message transmitted by the retrieve-common is indeed static in its header format, this theory is not possible. The message header for any individual message is transmitted with the identical header format. No header adjustments are made due to subsequent changes in the database size, since the subsequent data are transmitted in subsequent messages. When a block of records are required to be transmitted, the same header format for their addresses is used. Thus, the message header is constructed in the same fashion.

The next theory is whether or not the **BucketInfo message** can accommodate an excessive message size. The buffer for the **BucketInfo message** is filled with records by using a standard looping mechanism which contains a record counter, **K**. This record counter is used to keep track of the number of records inserted into the buffer. Additionally, a byte counter, **i**, is used to determine the length of all the records presented for transmission to other backends. This byte counter is used in conjunction with **K** to determine whether or not there is enough buffer space for the incoming records. If there is not, **BucketInfo message** is then transmitted to a exception procedure of the operating system.

The capability of retrieve-common to properly fit the incoming records into the message buffer, even though it has a fixed size of 1400 bytes, illustrates this implementation is database-size independent. We therefore discount the theory that the size of the message buffer in retrieve-common is implemented in a fashion that will allow the system to fail due to overloading of the buffer with a large number of records.

The third communication-based theory suggests a defect exists in the retrieve-common's utilization of the communication protocols supplied by the operating system. A brief explanation of the communication protocol is necessary. The operating system used by MBDS provides two different methods: the reliable and unreliable datagram. Stream communications are via sockets which are named locations in a process. When a process wants to send a message to another process, it refers to the name of the socket in the other process and transmits the message to the named socket. The operating system insures the

communication is reliable and error-free. This type of communication is one-to-one communication, i.e., from one computer to another computer.

Datagram communications allow a message to be transmitted from one process to several processes. This is known as one-to-many communications, i.e., broadcasting. However, the datagram communication is not reliable, i.e., occasionally one of receiving processes does not get the messages. Thus, it is unreliable broadcasting.

The method of communications in MBDS is reliable broadcasting based on the use of reliable sockets and unreliable datagrams for interprocess communication. A message is always broadcasted first via the datagram communication to all the other processes. If some receiving processes have not acknowledged the receipt of the message, the message is retransmitted to them via their sockets. A key aspect of this retransmission is that the socket names are never changed, and new sockets are not established during the retrieve-common. Thus, the broadcast mode of transmission in retrieve-common is reliable and fail-safe. The discounting of the last communication related-theory allows us to begin the evaluation of other theories.

2. Storage-Related Theories of Defects

To identify storage-related defects, we first review storage structures used in the testing of the retrieve-common. The first storage structure tested is the buffer structure in postprocessing. It may be implemented without the capability to handle large amounts of data. Additionally, it may not provide a unique buffer for the results of the retrieve-common. If these are indeed the cases, then they may indicate why the retrieve-common cannot output large amounts of data.

Our analysis has determined that there is only one designated output buffer for MBDS. Retrieve-common does not provide its own output buffer. We then direct our analysis to this buffer. The buffer is implemented as an array of characters with a maximum size of 1400 bytes. The procedure determines the amount of space available in the buffer and loads the empty space with records waiting to be output. To empty the buffer, the procedure passes the contents of the buffer via a message directly to the user-interface.

We also find that MBDS utilizes the same procedure, storage structure, and buffer to provide output to the user interface for all the other primary operations. This review invalidates our theory that either the storage structure of the postprocessing buffer or the procedure in postprocessing the buffered records is defective.

The conclusion that the output structure is implemented correctly has led us to review the correctness of input structures. Input structures deal with storage structures of data coming from secondary storage devices such as the paging disk. Retrieve-common requires that every record of the source file satisfied be entered into the virtual memory. If the size of the source file is large more virtual memory would be required. As with any secondary storage device, limitations do exist on the number of source records the paging disk may support. Also the paging disk is smaller than the base-data disk of a backend. The possibility of a paging-disk overflow is considered here. Additionally, this analysis allows us also to review the implementation of the input buffer. There may be a defect in the input buffer as well.

The new disk input and output (disk i/o) function is implemented to overload the paging disk by reaching the user's limit on base-data store known as

Quota; which contains allocated disk storage for the base-data of a particular user. The disk i/o function reads an entire track from the base-data disk into the **Track-Buffer. The Track-buffer** is implemented as an one dimensional array of 12,800 characters which is the size of a track also. When the disk read is completed, the contents of the **Track-buffer** are verified. To ensure records retrieved from the base-data disk do not exceed the capability of MBDS to process them, all of the contents in the **Track-buffer** are processed prior to reading another track of records. This processing consists of the verification of records based on the query and hashing the appropriate records into the virtual memory for later merging. In other words, this procedure ensures that the large amounts of data on the base-data disk do not overrun the buffer space. More importantly, the data can be processed one track at a time.

The ability to control input rates from the database stores has provided us with the evidence that the disk i/o process is not the cause of the system's defect. Therefore, we remove the disk-storage-related theory of defects from further consideration.

The final storage-related theory of defects to be reviewed is the theory of the virtual-memory inputs/outputs. Even though, the track-buffer and the disk i/o process ensure positive control of information input, they fail to account for information retrieved from other sources. Each backend has the capability to transmit a message up to 9200 bytes. To process the message, the backend must store it in the virtual memory which may overload the paging disk.

The virtual-memory i/o process is used in the retrieve-common. Its goal is to provide efficient temporary storage of records received from other

sources in the virtual memory. Our analysis is focused on the virtual memory i/o process.

a. Hashing and Storage of Records

The retrieve-common begins with TP, i.e., the Request Processing process. In this process, the type of query is identified, formatted, and transmitted to the backends. In Appendix C, we provide a review of the specific subprocedures involved in this process. The following high-level summary of procedures is provided prior to our determination of the problem.

The retrieve-common differs from the other primary operations after the disk i/o process is completed. The following steps of the retrieve-common operation also indicate where the difference occurs:

Step 1. Allocate space in the virtual memory to store information about the primary operation.

Step 2. The directory management process provides a list of addresses of tracks that contain records likely to satisfy the query. Each of these tracks is fetched from the base-data disk and placed into the virtual memory, i.e., the **Track-buffer**.

Step 3. The records in the track buffer are examined one record at a time. If the record is marked for deletion or does not satisfy the query, it will be discarded. If the record does satisfy the query, appropriate attribute values are extracted. The record is placed in an result buffer.

Step 4. This is where retrieve-common differs from all the other primary operations. When the result buffer is full, the extracted attribute values of records in the buffer are sent to a function **HashFunc,** which provides the virtual memory addresses and temporary storage of these records. This function is unique to the primary operation.

Step 5. Steps 2, 3 and 4 are repeated until all of the addresses provided by the directory management process are processed, the tracks at these

addresses accessed, and the records satisfying the query hashed into the virtual memory.

It is important to note that these five steps are designed for the source query. They are not duplicated for the target query; since records satisfying the target query, although hashed, are not stored temporarily in the virtual memory, i.e., records whose different attribute values are hashed into the same virtual memory address, as those in Step 4. Our analysis of the hashing function will begin in Step 4. The process of hashing records into the virtual memory requires the process to extract the common attribute value of a record from the result buffer, to develop a virtual memory address confined within the hashed address space, and to place the attribute value and record address in the hashing table. In addition to these capabilities, the process also resolves any collision. This ability is based on a chaining method where colliding records , i.e., records whose different attribute values are hashed into the same virtual memory address, are linked together.

In Appendix D, we provide an transaction flow of the steps involved in the determination of virtual memory addresses of records of the transaction. We only address those steps here where there are defects.

The original hashing algorithm is presented below:

Step 1: Extract the common attribute value (attr-value) from a record in the result buffer.

Step 2: If the syntactic type of **attr-value** is of the string type, then place the first two characters of **attr-value** in the temporary variables **c1** and **c2.** Otherwise, designate **attr-value** as a number, and assign to a temp variable.

Step 3: Calculate the bucket number. If **attr-value** is a string and the second character is < = 48 and = 0, the bucket number is (**c1** - 65) * 36. If **c2** > 48, the bucket number is ((**c1** -65) * 36) + (**c2** - 48). If **c2** > greater than but not equal to 48, then bucket number is calculated as ((**c1** - 65) * 36)+ (**c2** - 97) + 10.

Step 4. If **attr-value** is a small integer, 2, the bucket number would be attr-value - 0.

Step 5 If **attr-value** is a large integer, 3, the bucket number is (**attr-value** - 0) / .61

Step 6 This bucket number and record will be input into a temporary buffer and the common attribute of the next record is processed in Step 2.

The above algorithm failed to fulfill the two premises of hashing: randomness and uniformity [Ref. 8]. A good hashing function transforms a set of keys, i.e., common attribute values, to a set of random locations uniformly distributed in the range of hash table [Ref. 9].

The present hashing algorithm fails to randomly disperse records when the first two characters of the common attribute value are the same and of the string type. For example, given the following two customer codes, C102 and C103 as common attribute values, the algorithm will compute them as follows:

For C102, (67 -65) * 36 = 72 (bucket number)

For C103, (67 -65) * 36 = 72 (bucket number).

Each of them would furnish the same bucket number, i.e., virtual address, to place their respective records.

Although this example only shows the lack of randomness, the other deficiency, lack of uniformity, is illustrated by the way the algorithm uses a calculation that is different from the one used on string values. For example,

given the following two customer codes, 835 and 916 as common attribute values, the algorithm will compute them as follows:

For 835, 835 - 0 = 835 (bucket number).
For 916, 916 - 0 = 916 (bucket number).

Therefore, the determination of virtual addresses for records is based on two separate calculations.

The collision resolution technique is reviewed. The hashing function ensures that each of the 8192 buckets in the hash table serve as the head of a link list of blocks. When a block of the bucket has reached its limit of 1000 bytes, a operating-system call, alloc, is made for more memory in order to construct a new block. The new block is then filled with the waiting record. If the original block has not reached its capacity, the new record is inserted.

This type of collision handling is effective, if it is used in conjunction with a hashing function that ensured uniformity and randomness [Ref. 8]. The ideal uniformity will be that each link list of blocks has the same number of collided records. Additionally, the effective randomness will keep the number of collided records in the link list small. If an uniform distribution of records does occur, the hash table and the bucket size allows for approximately 245,000, 32-byte records to be stored before any collision takes place.

However, uniform distribution does not occur in most instances. The hashing function allows for the worst possible distribution to occur, i.e., the hashing of every common attribute value to the same bucket. Thus, the insertion or searching operations has the same level of performance as a linear search method which is inefficient for the hashing function.

b. Defects in Hashing

With the evidence that the hashing algorithm is defective, we then determine what is the impact on the MBDS system. We find the separate chaining technique in collision handling correlates with the message-header and buffer-error indicators received in our test runs. Also, we find that the time allocation is important to the well-being of the retrieve-common.

The collision handling using the separate chaining technique is noted for its capability to grow as a link list as long as needed. However, this growth is mediated by the memory availability. The capability of the present file system to provide the memory necessary to maintain the growth of the link list is questionable. The file system allows for the segmentation of memory into variable sizes [Ref. 7]. Additionally, the amount of memory allocated to a particular retrieve-common cannot be dynamically increased. Therefore, a very large set of records from both the source and target files can run out of memory.

The memory size for the buckets of a retrieve-common is too small. During an operational test that requires large sizes of data to be hashed into the virtual memory, a write error is observed. This error is a direct result of the fact that the retrieve-common has used up its allotted partition [Ref. 7]. Using software monitors, we dynamically observed the dedication of available memory to processes performing tasks for the retrieve-common. A utilization level of approximately 99 percent has been observed moments before the MBDS system is shut down.

With the evidence that the defective hashing algorithm is the cause of shutdown, we work to correct the defect. The revised hashing algorithm is

designed to provide randomness and uniformity which are lacking in the original algorithm.

D. A NEW HASHING ALGORITHM

We first ensure that the new algorithm is applicable for all possible key types, i.e., all possible value types of the common attribute.

The technique consists of transforming every character of the common attribute value to its internal representation i.e., an ASC II integer [Ref. 10]. The sum of all the characters of the common attribute values (called x) is now presented to the hashing function. An example of this new technique is illustrated below:

For C102, we have C = 67, 1 = 49, 0 = 48, and 2 = 50.

Thus, x = 67 + 49 + 48 + 50 = 214.

The randomness of our hashing function is provided by the division method [Ref. 7]. This method is defined as **H(x) = x mod m + 1**, where **m** is preferable a prime and **x** is the same as defined above. This computation basically provides the remainder of the division of **x** by **m**. The remainder plus one is the virtual-memory address.

The division method is used because it insures an address within the size, **m**, of the hashing table. Additionally, the division method ensures that if the table size is a large prime number, any collision of common attribute values is uncommon [Ref. 8]. For example, given **x** with a value of 214 and a hashing table whose size, m, is 8191 buckets, the following address calculation occurs:

$$H(x) = x \bmod m + 1$$

$$H(214) = 214 \bmod 8191 + 1$$

$$= 215.$$

The new hashing algorithm is presented below:

Step 1. Extract the common attribute value (**attr-value**) from the record in the result buffer.

Step 2. Transform each character of **attr-value** to its internal ASC-II representation.

Step 3. Calculate the sum (**temp**) of their ASC-II.

Step 4. Conduct the modulo division on **temp**. The resulting remainder plus one is the hashing-table entry.

Step 5. The record is directed to the virtual memory storage via the appropriate hashing-table entry.

The operational testing of the new hashing algorithm indicate that the hashing errors of the original algorithm have disappeared. In addition, the new hashing function provides variable buckets which are absent in the original function.

E. AN UNFORSEEN COMMUNICATION-RELATED DEFECT

An unforeseen error is discovered while conducting testing on the retrieve-common with large databases. This error is directly related to the operations of MBDS backends.

We recall the that retrieve-common requires each backend to transmit their target records to the other backends. A message transmission error occurs during this transmission. We observe that no error occurs if the message containes all of the records (i.e., not segmented). Additionally, if the portion of the message sent is the first segment of several message segments, the message is

error-free. An error occurs if the message has not met either of these two conditions.

The message error occurs only when the first 27 characters of the message body are incorrect. The attribute that is necessary to determine the virtual address of the record is therefore incorrect. As a result value, the hashing function attempts to compute an virtual address using an incorrect value. Incidentally, the value that the hashing function used is always the content of a register used in an earlier operation. The effect of using 16 characters to compute the virtual address has led to an address too large for the operating system to handle. This excessively large address caused a core dump and immediate system shutdown.

Our analysis shows that message timing is the cause of the message-error. This conclusion is based on an exhaustive analysis of a sample bucket-message traffic during different phases of transmission. The bucket message is reviewed (1) before and after transmission between processes in the same backend, (2) prior to being inserted into the operating system for interprocess communication among backends via the interprocess communication (ip) buffer, and (3) after the receipt by the backends. The bucket message is correct in all three locations except when it is placed in the ip buffer of the operating system. The ip buffer is an intermediate buffer of the operating system for message transmission [Ref. 11]. However, though the message goes into the ip buffer correctly, it exits incorrectly.

The ip buffer has a size of 1000 bytes [Ref. 11]. But, the size of the messages to be inserted into this buffer is up to 1425 characters. With the size of the message larger than the buffer size, we discover that a flushing mechanism is

used. It ensures that as the buffer reaches it limit, it first outputs its contents to the appropriate source and then allows the receipt of additional messages. Our tests indicate this mechanism has not been given enough time to complete the flushing task. When the number of target records to be transmitted require multiple bucket messages, the messages are damaged in the ip buffer.

The size limitation of the ip buffer and its slow performance when transmitting multiple target records point to a message-timing error. The input speed of messages entering the ip buffer is faster than the speed that the ip buffer can empty its contents by sending out as a message. These differences in capabilities cause the messages in the buffer to be affected by incoming records. One expedient way to overcome this limitation is to allow enough time for the flushing mechanism to complete each flushing task.

IV. A SUMMARY OF FINDINGS

A. DEFECTS DISCOVERED

The retrieve-common operation has not been performing correctly due to a communication-related timing defects and a defective hashing function.

1. Causes of the Communication-Related Defects

The communication-related defects have been caused by a buffer-timing error. The operating system's communication buffer is unable to completely flush its contents before the arrival of the next message. Therefore, in some instances, the contents of the communication buff - can be inadvertantly modified which provides the neccessary conditions for .he ioctrl error.

2. The Defects of The Hashing Function

The hashing function is considered defective because it fails to provide randomness and uniformity. In the case of randomness, when the first two letters of the common-attribute value are the same, the hashing function generates the same virtual address. The lack of uniformity is evident when different address calculations are used for string and numerical attribute values.

The defect in the hashing algorithm is apparent when we use large databases which assign records to the same virtual address. The hashing function exhausts the user's memory allotment which leads to the write error.

3. Other Findings Concerning Defects

The cause of the bus error that we observed during our theorizing stage is now known. Since MBDS is a loosely coupled system, the backends' operating systems work independently. When an abnormal termination occurs on one

backend, it does not automatically cause the termination of the other backends. Processes which are interacting with the backend that terminated may shutdown, but the others will not shutdown. These remaining processes require manual termination. This need for manual termination can result in the occurence of duplicate processes if MBDS is reactivated.

MBDS does not allow duplicate processes. Therefore, the operating system presents a bus error when the MBDS system is re-activated and duplicate processes exist. This deficiency is corrected by developing a program which will shutdown all processes on the backends prior to MBDS reactivation.

B. BENEFITS OF THIS RESEARCH PROJECT

The benefits of this research are substantial. They are presented below:

a. We have determined that the MBDS process architecture is effective. The location of the merging functions takes advantage of the peculiarities of the system network and minimizes delays.

b. We have developed and presented a documentation structure that will assist system designers and maintenance staff to design and service complicated software. Examples of such documentation are presented in appendices.

c. We have presented a methodology for efficient trouble-shooting of complex parallel-software systems. With the increasing development of parallel systems, this methodology provides an effective guide to system staff who conduct system maintenance.

d. We have determined the causes of the defective performance of the Primary Operation , Retrieve Common. We are able to correct one of the defects; the problematic hashing algorithm. However, the communication-timing defect will require further analysis. The timing analysis necessary to flush the ip buffer is beyond the scope of this study, besides, it is a problem inherited in the operating system, not the MBDS system.

e. Finally, we have corrected the file-path errors which adversely affect the ability to develop test databases.

The end result of this research is that the Primary Operation, Retrieve Common that can now manipulate and merge a database 500% larger than at the outset of this research. More importantly, we have provided an outline for the successful trouble-shooting of complex parallel systems.

C. FUTURE WORK

The next step in the development of the MBDS system is to correct the communication-related timing defect as indicated in item 4 of the previous section. This may require some modifications of the operating system, i.e., Berkely 4.3 Unix.

APPENDIX A. RECORD PROCESSING MAP

This documentation is a highlevel presentation of functions which exist within the RECP process. The documentation provides information on functions within the process, their basic capabilities, and the file where the function is defined. This documentation will provide a quick reference guide to staff and experienced users.

```
--------                        ------- -------
FUNCTION                        SRC(.c) PURPOSE
--------                        ------- -------
main                            recproc

RecP_init                       recproc initialize
  initsr                        sndrcv  initialize communication channels
    <>
  disk_init                     disks   initialize disk i/o
    <>
Msg$RP_R                        recpsr  get the next message
  chk_waiting_req               chkwait is request waiting for region?
    <>
  putRid                        recpsr  put request id in message buffer
    <>
  receive                       sndrcv  receive a message
    <>
  wait_msg                      waitmsg wait for message or I/O completion
    <>
Sender$RP_R                     recpsr. get the sender
  <>
################################################################################
|RP_DM                          recproc message from DM
  Type$RP_R                     recpsr  get the message type
    <>
  ============================================================================
  |ReqProcessing                recproc process a request
    ReqAddrs$RP_R               recpsr  return request in buffer
      <>
    get_tmpl_ptr                dbtmpmod  get ptr to record template
      <>
    RB$GET                      rbabs   allocate a result buffer
      <>
    ALL_STO_RP_ri               allsto  allocate storage for request
      Check_for_By              allsto  allocate hash info structure
        <>
      aggr_op                   allsto  finds any agg op in request table
        <>
    Rid$RP_R                    recpsr  get the request id
      <>
    |ST_Insert                  stins   case INSERT
      TB_FETCH                  disks   fetch a track buffer for insertion
        get_free_dio_reg        disks   get a region
          <>
        put_info_dio_reg        disks   put information in the region
          <>
        map_dio_reg             disks   map to the region
          find_dio_reg          disks   get index of dio entry
            <>
          map_TB                unixdisks  set the TB ptr
            <>
        Dio$RP_S                recpsr  send I/O message to DIO
          send
            <>
```

```
    get_free_dio_reg          disks   get a region
      <>
    put_info_dio_reg          disks   put information in the region
      <>
    map_dio_reg               disks   map to the region
      <> as above
    $INS_PROCESSING           insp    insert a record
      $IP_INSERT_RECORD       insp    insert the record into the track buffer
        <>
      TB_STORE                disks   store track buffer back to the disk
        find_dio_reg          disks   get index of dio entry
          <>
        map_dio_reg           disks   map to the region
          <> as above
        Dio$RP_S              recpsr  send I/O message to DIO
          send
            <>
  |ST_RetDel                  stretdel  case DELETE
    TB_FETCH                  disks   fetch a track buffer for insertion
      <> as above
    RB$SEND_COMPLETION        rbabs   send completion signal to controller, CC
      HASH_FUNC               retcom
        Broadcast_Target_Info retcom
          send
            <>
        HASH_THE_RECORD       retcom
          PutHashBuffer       retcom
            BucketBlock       retcom
              StoreRecord     retcom
                AllocBlock    retcom  allocate a block
                  <>
          Broadcast_Target_Info  retcom
            send
              <>
        MERGE                 retcom
          RES_CNTL$RP_S       recpsr  send the results to the controller
            send
              <>
      RES_CNTL$RP_S           recpsr  send the results to the controller
        send
          <>
      DM_FinReq$RP_S          recpsr  send the request id (update) to DM
        <>
      CC_FinReq$RP_S          recpsr  send the request id (non-update) to CC
        putRid                recpsr  put request id in message buffer
          <>
  |ST_Update                  stupd   case UPDATE
    TB_FETCH                  disks   fetch a track buffer for insertion
      <> as above
    ReqP_NoMoreGenIns$RP_S    recpsr  send message to REQP
      send
        <>
    RP_ContinueGenIns         rpcont  INSERTs caused by an UPDATE can continue
      <>
    RB$SEND_COMPLETION        rbabs   send completion signal to controller, CC
      <> as above
==========================================================================
|Changed_ClusRes              changed a record has changed cluster
  RhccAns$RP_R                recpsr  receive DM's answer on cluster change
    <>
  RPUPD2                      updp
    map_dio_reg               disks   map to the region
      <> as above
    RHCC$RP_S                 recpsr  send the new record to REQP
```

```
      send
        <>
    TB_STORE                    disks   store track buffer back to the disk
      <> as above
  ================================================================================
  |No_MoreGenIns                nomore  no more generated inserts for an UPDATE
    Rid$RP_R                    recpsr  get the request id
      <>
    RB$SEND_COMPLETION          rbabs   send completion signal to controller, DM
      <> as above
################################################################################
|RP_RP                          recproc "message" from self
  Type$RP_R                     recpsr  get the message type
    <>
  ================================================================================
  |ReqProcessing
    <> as above
################################################################################
|RP_CNTL_ANOTHER_BE_MSG         recproc message from TI
  Type$RP_R                     recpsr  get the message type
    <>
  ================================================================================
  |Common messages              commsg  see commsg.map
    <>
  ================================================================================
  |Rid$RP_R                     recpsr  retrieve common - allocate space
     <>
   All_Sto_RP_ri_RetCom         allsto  allocate structure space
     <>
  ================================================================================
  |Msg_q$RP_R                   recpsr  set ptr to next msg in queue
    <>
   PROCESS_BE_Target            retcom
     StoreRecord                retcom
       AllocBlock               retcom  allocate a block
         <>
     MERGE                      retcom
       RES_CNTL$RP_S            recpsr  send the results to the controller
         send
           <>
  ================================================================================
  |DioStop$RP_S                 recpsr  send a stop message to DIO
    send
      <>
################################################################################
|RP_DIO                         recproc message from DIO
  Type$RP_R                     recpsr  get the message type
    <>
  ================================================================================
  |RP_WriteCompleted            recproc physical write is completed
    RidAddr$RP_R                recpsr  get request id of completed read
      <>
    |WC_Insert                  wcreqs  if INSERT
      RB$SEND_COMPLETION        rbabs   send completion signal to controller, CC
        <> as above
      CC_FinReq$RP_S            recpsr  send the request id (non-update) to CC
        putRid                  recpsr  put request id in message buffer
          <>
      RecP_free                 rpfree  free the space used by a request
        <>
      set_free_dio_reg          disks   find entry for a request
        find_dio_reg            disks   get index of dio entry
          <>
    |WC_Delete                  wcreqs  if DELETE
```

```
    RB$SEND_COMPLETION          rbabs   send completion signal to controller, CC
      <> as above

  |WC_Update                    wcreqs  if UPDATE
    ReqP_NoMoreGenIns$RP_S      recpsr  send message to REQP
      send
        <>
    RP_ContinueGenIns           rpcont  INSERTs caused by an UPDATE can continue
      <>
    RB$SEND_COMPLETION          rbabs   send completion signal to controller, CC
      <> as above
    set_free_dio_reg            disks   find entry for a request
      find_dio_reg              disks   get index of dio entry
        <>
================================================================================
|ResData$RP_R                   recpsr  restore data received from DIO
  find_dio_reg                  disks   get index of dio entry
    <>
  get_TBptr                     unixdisks  get ptr to track buffer
    <>
|RP_ReadCompleted               recproc physical read is completed
  RidAddr$RP_R                  recpsr  get request id of completed read
    <>
  |RC_Insert                    rcproc  if INSERT
    map_dio_reg                 disks   map to the region
      <> as above
    $INS_PROCESSING             insp    insert a record
      <> as above
  |RC_Ret                       rcreqs  if RETREIVE[-COMMON]
    TB_FETCH                    disks   fetch a track buffer for insertion
      <> as above
    $RETR_PROCESSING            retp    process RETREIVE
      map_dio_reg               disks   map to the region
        <> as above
      CHK_QUERY                 chkqry  check whether record satisfies QUERY
        <>
      RP_aggregate              retp    calculate any aggregate operations
        XTRACT                  retp    get attribute and value for target list
          <>
        BY_HASH_FUNC            retby
          BY_HASH_RECORD        retby   hash and store the records
            StoreByRecord       retby
              AllocByBlock      retby   add a new bucket to the end of the list
                <>
        RB$AG_PUT_SEND          rbabs   put aggregate results into result buffer
          RB$PUT_SEND           rbabs   put request results into result buffer
            <> as above
        fill_res_buff           retp    fill result buffer
          <>
      XTRACT                    retp    get attribute and value for target list
        <>
      BY_HASH_FUNC              retby
        <> as above
      RB$PUT_SEND               rbabs   put request results into result buffer
        HASH_FUNC               retcom
          <> as above
        RES_CNTL$RP_S recpsr    send the results to the controller
          send
            <>
      Send_Hash_Info            retby
        RB$PUT_SEND             rbabs   put request results into result buffer
          <> as above
      RB$AG_PUT_SEND            rbabs   put aggregate results into result buffer
        RB$PUT_SEND             rbabs   put request results into result buffer
```

```
          <> as above
        set_free_dio_reg        disks   find entry for a request
          find_dio_reg          disks   get index of dio entry
            <>
      RB$SEND_COMPLETION        rbabs   send completion signal to controller, CC
        <> as above
      free_bucket               retby   free the space used by a block
        free_bucket
          <>
      RecP_free                 rpfree  free the space used by a request
        <>
    |RC_Delete                  rcreqs  if DELETE
      TB_FETCH                  disks   fetch a track buffer for insertion
        <> as above
      $DEL_PROCESSING           delp    process DELETE
        map_dio_reg             disks   map to the region
          <> as above
        CHK_QUERY               chkqry  check whether record satisfies QUERY
          <>
        TB_STORE                disks   store track buffer back to the disk
          <> as above
      RB$SEND_COMPLETION        rbabs   send completion signal to controller, CC
        <> as above
      set_free_dio_reg          disks   find entry for a request
        find_dio_reg            disks   get index of dio entry
          <>
    |RC_Update                  rcreqs  if UPDATE
      TB_FETCH                  disks   fetch a track buffer for insertion
        <> as above
      $UPD_PROCESSING           updp    process UPDATE
        map_dio_reg             disks   map to the region
          <> as above
        CHK_QUERY               chkqry  check whether record satisfies QUERY
          <>
        INC_URCPT               updp    increment records being updated
          <>
        $UPD_RECORD             updp    UPDATE the record
          <>
        ONV$RP_S                recpsr  ask DM whether record changes cluster
          send
            <>
      ReqP_NoMoreGenIns$RP_S    recpsr  send message to REQP
        send
          <>
      RP_ContinueGenIns         rpccnt  INSERTs caused by an UPDATE can continue
        <>
      RB$SEND_COMPLETION        rbabs   send completion signal to controller, CC
        <> as above
      set_free_dio_reg          disks   find entry for a request
        find_dio_reg            disks   get index of dio entry
          <>
################################################################################
|RP_shutdown                    recproc shutdown process
  finishsr                      sndrcv  finish send/receive
    <>


COMMON FUNCTIONS
----------------
FIND_RP_ri                      findrp  get ptr to request info structure
  <>
```

APPENDIX B. RECORD PROCESSING PSEUDO-CODE

This documentation is a midlevel presentation of events occuring within the RECP process. The intent is to provide the user with a basic understanding of the activity that occurs during specific events. It does not represent the exact steps taken within a function.

```
External Variables
------------------
struct tb_info       dio_reg[MAX_DIO_REG]
struct RP_rid_info   *front_RP_rid_info
struct RP_rid_info   *rear_RP_rid_info
char                 *TB

Pseudo Code
-----------
Initialize process (RecP_init in recproc.c)
  Initialize communication channels (initsr in sndrcv.c)
  Initialize variables related to disks (disk_init in disks.c)
    Set up the track buffers for each region used by disk I/O
      Set dio_reg[DIO_REG].ti_reg_status = REG_FREE (not being used)
Set StopSys = FALSE
Enter message receiving loop; continue while StopSys = FALSE
  Get the next message (Msg$RP_R in recpsr.c)
    Check if any request is waiting for a region (chk_waiting_req in chkwait.c)
      Traverse linked list of struct RP_rid_info's to check whether any has
        RP_ri_status of WAITING
    If a request is waiting for a region
      Put traffic id and request number into message buffer
      Fill message header with sender and receiver equal to RECP; and type
        equal to OLD_REQ
      Return
    Else if no request is waiting for a region
      Check to see if there is a new message (receive)
        Wait flag is TRUE
        If there is a message return
    Wait for a message or an I/O completion (wait_msg in waitmsg.c)
      [Can this function be reached?]
  Get the sender name of the message (Sender$RP_R)
  Switch on message sender
################################################################################
case DM (RP_DM)
  Get the type of the message (Type$RP_R in recpsr.c)
  Switch on message type
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case ReqDiskAddrs (ReqProcessing in recproc.c)
    Get the request (ReqAddrs$RP_R in recpsr.c)
      Copy the database id into dbid[]
      Copy the request into the request table (request->req_tbl)
      Copy number of addresses into addrs->as_no_addrs
      Copy each disk, cylinder, track no set into addrs->as_addrs[n]
      Copy new track flag to NewTrack
    Copy traffic id and request number from request table into struct ReqId
    If INSERT set tmpl_index = 7 else set tmpl_index = 8
    Get ptr (tmpl_ptr) to struct rtemp_definition
    Get ptr (RP_rb) to a result buffer structure (RB$GET in rbabs.c)
      Copy traffic id and request number from rid into request buffer
      Set RB_next_empty_pos = 0
```

```
Get ptr (RP_ri_ptr) to struct RP_rid_info (the main struct for the process)
  (ALL_STO_RP_ri in allsto.c)
  If RETRIEVE-COMMON
    ...
  If not RETRIEVE-COMMON
    Allocate space for the new RP_rid_info
    Link to list of RP_rid_info; set front_RP_rid_info and rear_RP_rid_info
    Copy traffic id and request number from rid into RP_ri_rid
    Set ptrs to NULL (RP_ri_hash, RP_by_hash, RP_agg_ptr)
    Set SrceDone = FALSE
  Copy the database id from dbid[] into RP_ri_dbid
  Copy the request into RP_ri_dbid
  Copy address set (disk,cylinder,track) into RP_ri_dbid
  If RETRIEVE
    ...
  If not RETRIEVE
    Set ptr in RP_ri_dbid to aggregate_info to NULL
  Set address of the index to be read (addr_ind) to 0
  Link rtemp_definition to RP_rid_info
  Link ResultBuffer to RP_rid_info
  Fill RP_ri_urcpt[] in RP_rid_info with 0's
  If not UPDATE caused by INSERT
    Set RP_ri_status in RP_rid_info to NOT_WAITING
  If UPDATE caused by INSERT
    ...
  Set RP_ri_no_completed_writes = 0
  Set this_BE_to_ins_count = 0
  Set no_more_gen_ins_msg_rcv = FALSE
If UPDATE caused by INSERT (RP_ri_status == UpdFirstPhaseWaiting)
  Return
Set req_type from req_tbl
If INSERT (ST_Insert in stins.c)
  If inserting a record into an old track (NewTrack == FALSE)
    ...
  If inserting a record into a new track (NewTrack == TRUE)
    Look for a free region (get_free_dio_reg in disks.c)
      Find 1st entry in global dio_reg array with ti_reg_status == REG_FREE
      If found set its ti_reg_status = REG_IN_USE
    If free region found
      Put information in the region (put_info_dio_reg in disks.c)
        Fill in traffic id and request number
        Fill in disk, cylinder, and track numbers
      Find entry and map to the region (map_dio_reg in disks.c)
        Find the entry for a request (find_dio_reg in disks.c)
          Match request and storage info to dio_reg elements until found
          Return index to entry (ind_dio_reg)
        Map to the region (map_TB in unixdisks.c)
          Set track buffer (TB) to tb entry corresponding to tb_info entry
      Set the beginning of each record sized division to no_rec ('3')
      Set the end of the buffer to EOTrack ('&')
      Issue the write ($INS_PROCESSING in insp.c)
        Get ptr (RP_ri_ptr) to the RP_rid_info entry (FIND_RP_ri)
        Insert the record into track buffer ($IP_INSERT_RECORD in insp.c)
          Scan track buffer to find the first free slot to insert the record
          If found
            Set first byte to rec_exist ('1')
            Set ptr (ptr) to next byte
            For each attribute
              Write value followed by EOField ('$')
            Fill in EORecord ('#')
          Record will be, for example: 1value1$value2$value3$#
        Unmap from the region (umap_dio_reg in disks.c)
          Free the TB so it does not point anywhere (umap_TB in unixdisks.c)
            Set TB to NULL
```

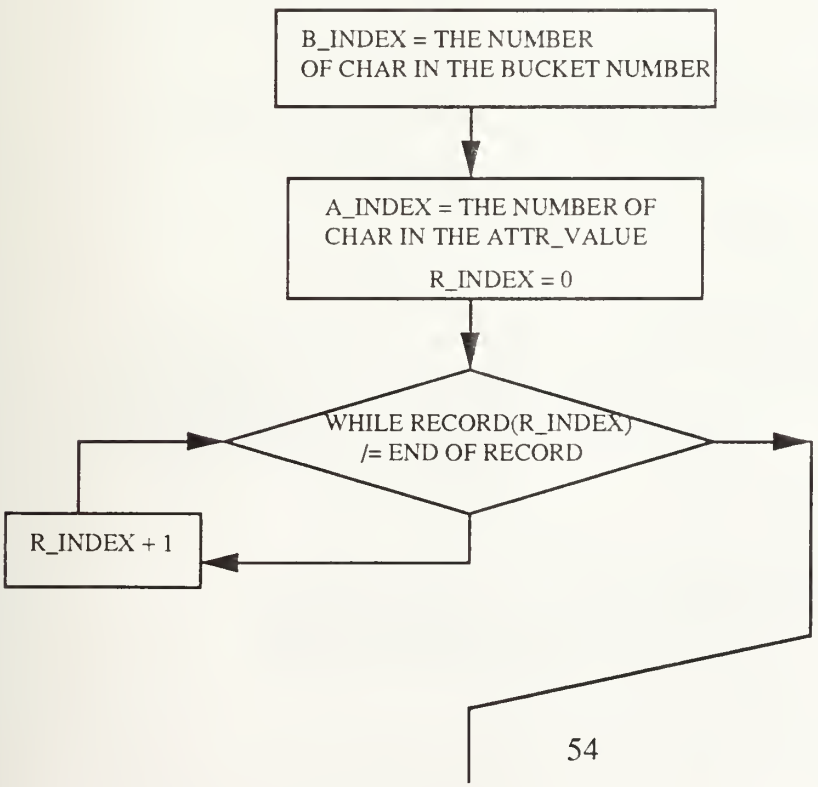
```
            Store TRACK_BUFFER back to the disk according to addr (TB_STORE)
              Find the entry for a request (find_dio_reg in disks.c)(as above)
              Find entry; map to the region (map_dio_reg in disks.c)(as above)
                TB points to the region
              Send the info to DISK I/O (Dio$RP_S in recpsr.c)
                Send request identifiers and contents of track
              Set the ti_reg_status for the region to REG_WRITE
              Unmap from the region (umap_dio_reg in disks.c)(as above)
          If free region not found
            Set RP_ri_status to WAITING
      If RETRIEVE, RETRIEVE-COMMON, DELETE (ST_RetDel in stretdel.c)
        ...
      If UPDATE (ST_Update in stupd.c)
        ...

  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case ChangedClusRes (Changed_ClusRes in changed.c)
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case NoMoreGenIns (No_MoreGenIns nomore.c)
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case Fetch
    <<< to be coded >>>
################################################################################
case RECP (RP_RP)
  Message from 'self'; a backlogged request is processed; no actual message is
    received
  Get the type of the message (Type$RP_R in recpsr.c)
  Switch on message type
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case OLD_REQ (ReqProcessing in recproc.c)
################################################################################
case G_PCLB (RP_CNTL_ANOTHER_BE_MSG in recproc.c)
  Get the type of the message (Type$RP_R in recpsr.c)
  Switch on message type
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  Common messages
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case RetComNotification
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case BucketInfo
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case Stop
################################################################################
case DIO (RP_DIO)
  Get the type of the message (Type$RP_R in recpsr.c)
  Switch on message type
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case PIO_WRITE (RP_WriteCompleted in recproc.c)
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  case PIO_READ
    Restore data from message buffer to track buffer (ResData$RP_R in recpsr.c)
    A physical read is completed (RP_ReadCompleted in recproc.c)
################################################################################
Shutdown process (RP_shutdown in recproc.c)
  Finish send/receive (finishsr in sndrcv.c)
```

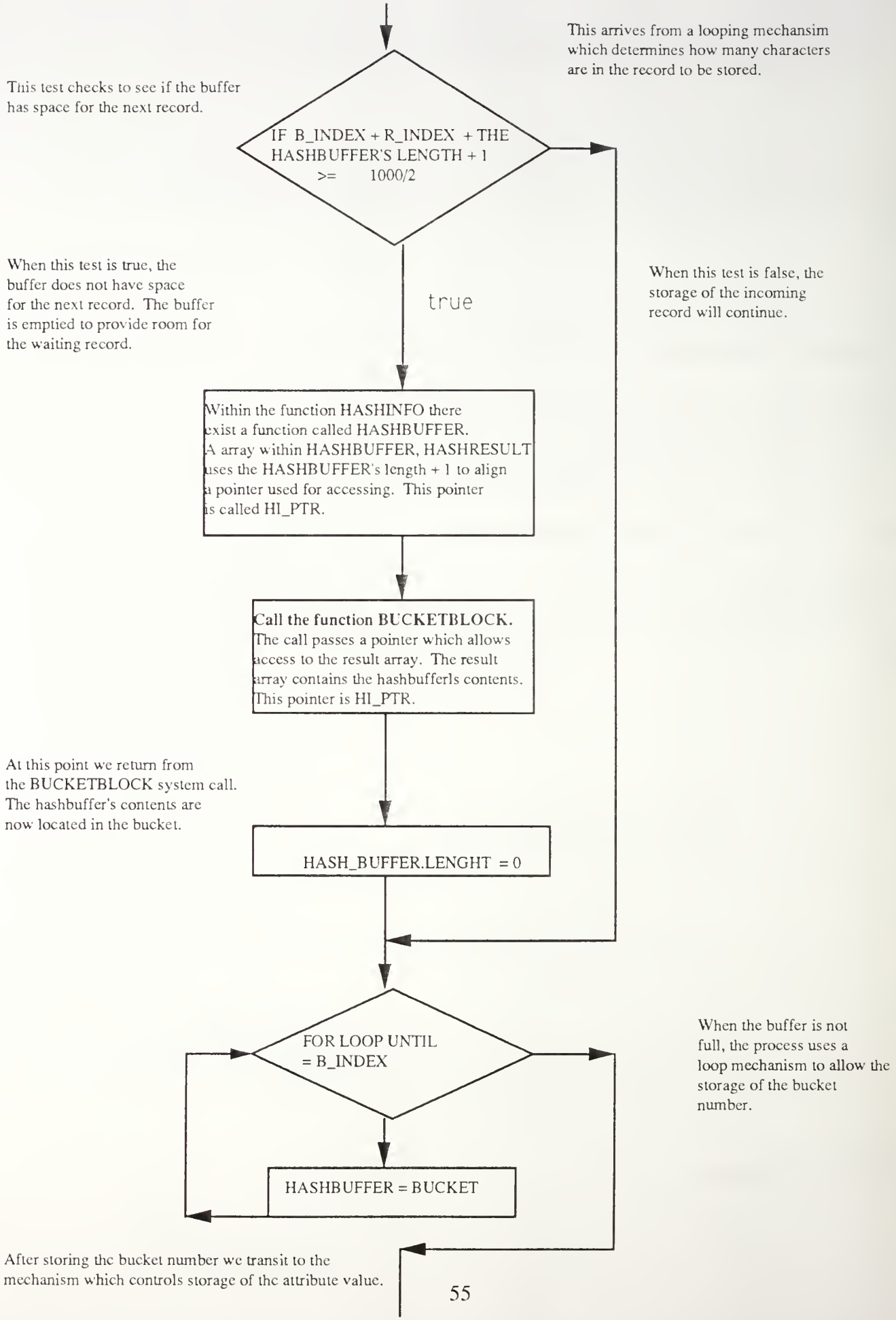
This documentation is a low-level presentation of the specific events occurring within the PUTHASHBUFFER function of RECP. It provides the function's name, a short description of variables passed in, and a logical flow of events.

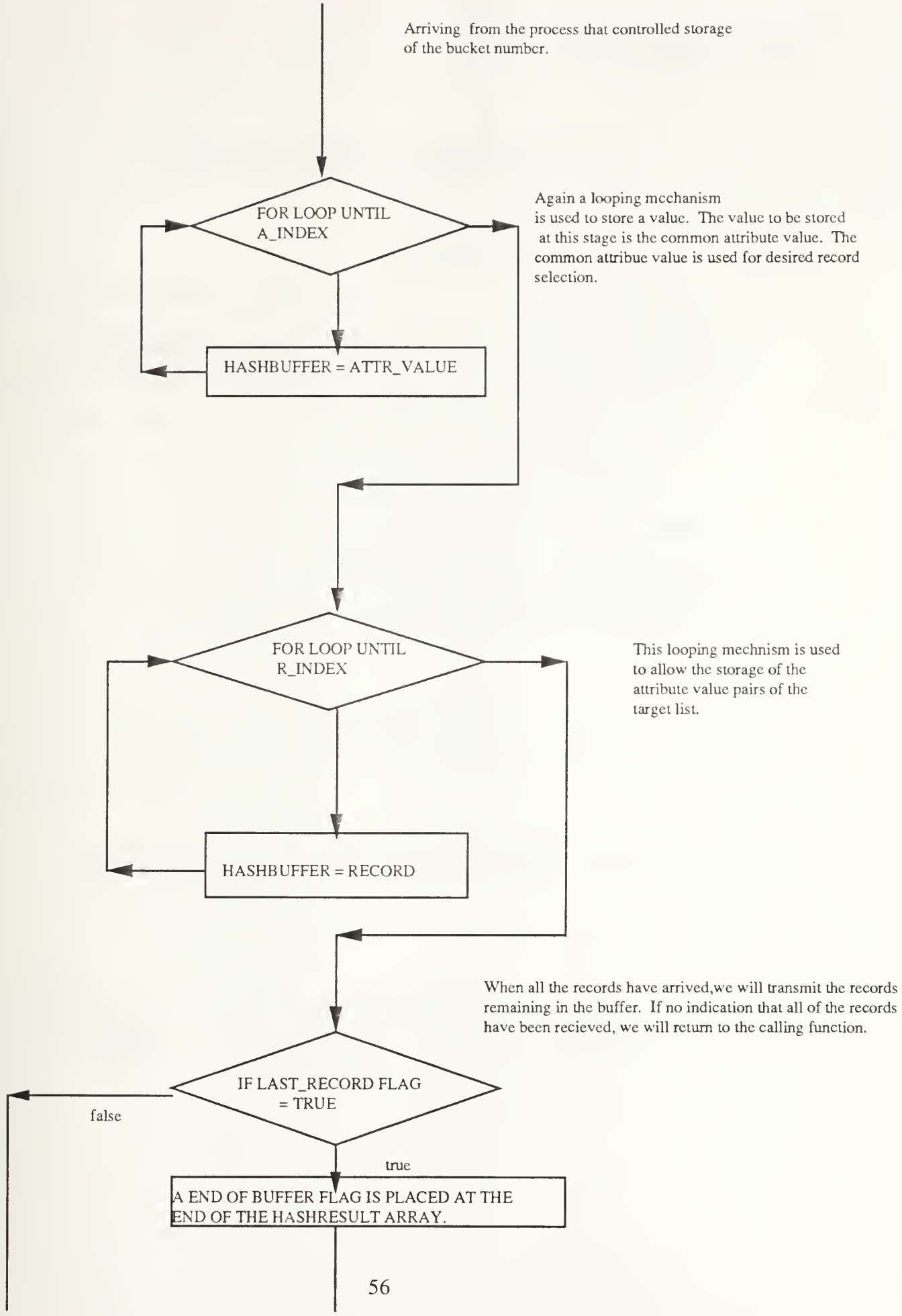
Function Name: PUTHASHBUFFER

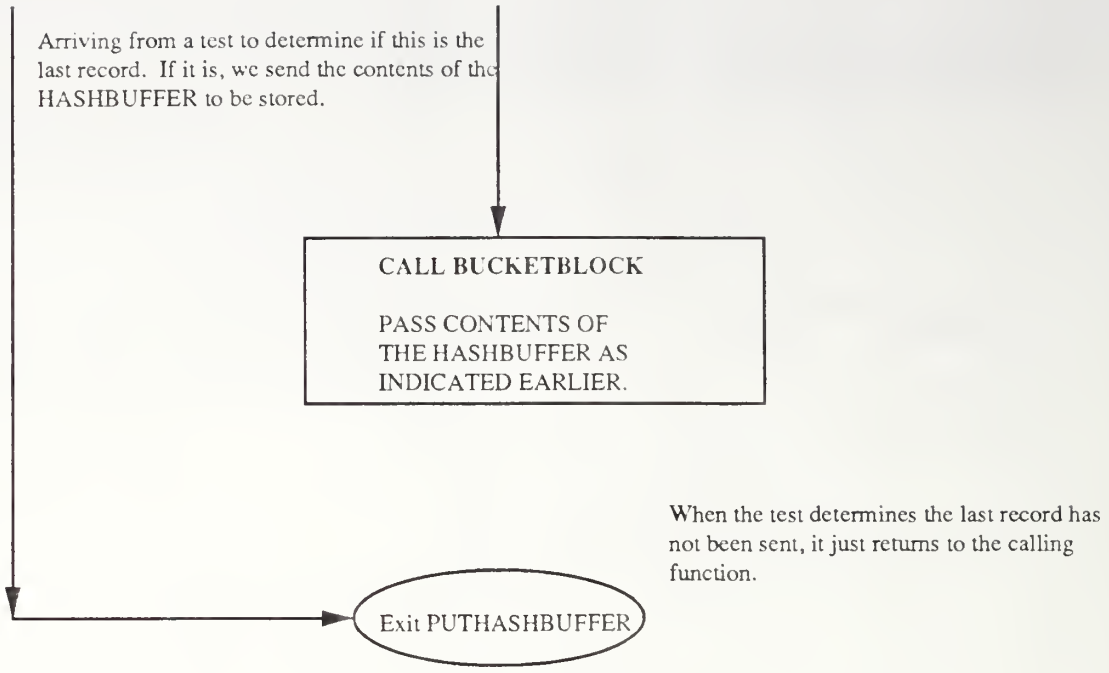
The following variables are passed in:

1. hi_ptr : This variable points at the function hashinfo. The function hashinfo stores the intermediate results of a retrieve common.
2. bucket: This is the virtual storage address; the bucket number.
3. attr_value: This is the specific attribute value of the query.
4. record : This is the contents of the result buffer after the attribute name and the attribute value has been extracted.
5. last record: This flag indicates whether a particular record is the last from a specific backend.




This arrives from a looping mechansim which determines how many characters are in the record to be stored.
This test checks to see if the buffer has space for the next record.
IF B_INDEX + R_INDEX + THE HASHBUFFER'S LENGTH + 1 >= 1000/2
When this test is true, the buffer does not have space for the next record. The buffer is emptied to provide room for the waiting record.
true
When this test is false, the storage of the incoming record will continue.
Within the function HASHINFO there exist a function called HASHBUFFER. A array within HASHBUFFER, HASHRESULT uses the HASHBUFFER's length + 1 to align a pointer used for accessing. This pointer is called HI_PTR.
Call the function BUCKETBLOCK. The call passes a pointer which allows access to the result array. The result array contains the hashbufferls contents. This pointer is HI_PTR.
At this point we return from the BUCKETBLOCK system call. The hashbuffer's contents are now located in the bucket.
HASH_BUFFER.LENGHT = 0
FOR LOOP UNTIL = B_INDEX
When the buffer is not full, the process uses a loop mechanism to allow the storage of the bucket number.
HASHBUFFER = BUCKET
After storing the bucket number we transit to the mechanism which controls storage of the attribute value.


Arriving from the process that controlled storage of the bucket number.
FOR LOOP UNTIL A_INDEX
HASHBUFFER = ATTR_VALUE
Again a looping mechanism is used to store a value. The value to be stored at this stage is the common attribute value. The common attribue value is used for desired record selection.
FOR LOOP UNTIL R_INDEX
HASHBUFFER = RECORD
This looping mechnism is used to allow the storage of the attribute value pairs of the target list.
When all the records have arrived,we will transmit the records remaining in the buffer. If no indication that all of the records have been recieved, we will return to the calling function.
IF LAST_RECORD FLAG = TRUE
false
true
A END OF BUFFER FLAG IS PLACED AT THE END OF THE HASHRESULT ARRAY.


Arriving from a test to determine if this is the last record. If it is, we send the contents of the HASHBUFFER to be stored.
CALL BUCKETBLOCK
PASS CONTENTS OF THE HASHBUFFER AS INDICATED EARLIER.
When the test determines the last record has not been sent, it just returns to the calling function.
Exit PUTHASHBUFFER

APPENDIX D. GUIDE TO MESSAGE ENTRIES

A. MESSAGE FORMAT INFORMATION

This appendix contains the format of all messages utilized on MBDS. Additionally, an example of the format of a Bucket Info message is provided. The message format that is used within MBDS is illustrated below:

Type: [message type]: This is represented by a 3 digit number.

Sender: [sending process(es)]: This is represented by a 3 digit number.

Reciever: [receiving process (es)] This is represented by a 3 digit number. One special note; if a Put is the reciever, the message is relayed to the Get in another machine.The ultimate reciever of the messages is indicated.

A BucketInfo message is presented below to illustrate the placement of the above format information.

```
..1..    ..3..
501504252XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX&
  2                                  4
```

1: Sender = RECP
2: Reciever = P_PCLB (all other backends)
3: Message type = BUCKET INFO
4: Message body = The message body will contained target records of a retrieve-common

REFERENCES

1. Lazou,C., *Supercomputers and their Use*, pp 2-20, Oxford Science Publications , New York 1988.

2. Elmasri, R., Shamkant B..N., *Fundamentals of DataBase Systems*, pp 556-565, Benjamin/Cummings, California 1989.

3. Galovich, S., *Introduction to Mathematical Structures*, pp 168-179, Harcourt Brace Jovanovich , San Diego 1989

4. Demurjian, S. A., Hsiao D. K., Marshall R. G., *Design Analysis And Performance Evaluation Methodologies For Database Computers*, pp 96-111, Prentice-Hall Englewood Cliffs, 1987.

5. Menon M.J. and Hsiao D. K., *Design and Analysis of Join Operations of Database Machines*, pp 203-226, Englewood Cliff, New Jersey, 1983.

6. Hsiao, D. K., "A Parallel, Scalable, Microprocessor-Based Database Computer for Performance gains and Capacity Growth," *IEEE Micro*, v , no 1, Jan 1992, pp 2-22.

7. Leffer S., McKusic M., Karels M., Quarterman J., *The Design and Implementation of the 4.3 BSD Unix Operating System*, pp 187-221, Addision Weseley, California, 1989.

8. Tremblay, J. P., *An Introduction to Data Structures With Applications*, pp 100-200, McGraw-Hill, New York, 1984.

9. Mander,U., *Introduction to Algorithms*, pp 78-80, Addison Weseley, California, 1989.

10. Skansholm, J., *Ada From the Beginning*, p 80, Addision Weseley, Amsterdam , 1989.

11 Sechrest, S., *Interprocess Communication Tutorial*, pp 1-10, Usenix California ,1986.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Chairman, Code CS Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Curriculum Officer, Code 37 Computer Technology Program Naval Postgraduate School Monterey, California 93943-5100	2
5.	Professor David K. Hsiao, Code CS/HQ Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
6	Gregory A. Hammond 1320 Spruance Rd, Monterey Ca 93940	3

Thesis
H17526 Hammond
c.1 The instrumentation of a parallel, distributed database operation, retrieve-common, for merging two large sets of records.